Summary of University of Idaho RISCV ISA Plans

Center for Secure and Dependable Systems

University of Idaho

October 2024

Jim Alves-Foss

# Overview

The intent of the new ISA model for RISCV is to support the concept of typed assembly language. The details and model of for that are available in a companion document. This document is meant to be a summary of the modifications and additions to traditional RISCV.

## Base ISA

We enhance RISCV based on the 20240411 Specifciations available from riscv.org

* Volume 1, Unprivileged Specification version 20240411  [[PDF](https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view?usp=drive_link)][[GitHub](https://github.com/riscv/riscv-isa-manual/releases/tag/20240411)]
* Volume 2, Privileged Specification version 20240411  [[PDF](https://drive.google.com/file/d/17GeetSnT5wW3xNuAHI95-SI1gPGd5sJ_/view?usp=drive_link)][[GitHub](https://github.com/riscv/riscv-isa-manual/releases/tag/20240411)]

Specifications can be found here: <https://github.com/riscv>. A subset of these is the Non-ISA specifications which will help with ELF and other information, specificationlly the

* [RSICV ABI Version 1.0](https://github.com/riscv-non-isa/riscv-elf-psabi-doc/releases/download/v1.0/riscv-abi.pdf)

With respect to the standard, we will start with

* RV32I Base Integer Instruction Set Version 2.1.
* Zicsr Extension for Control and Status Register (CSR) Instructions, Version 2.0
* M extension for Integer Multiplication and Division Version 2.0

### Bit Width

For the purposes of this model, we will use a 32-bit processor model.

### Registers

We support that standard 32-bit register sets as defined in RV32I and M extensions.

X0 – X32 as general-purpose use registers. Our assumption will be that these will be used with respect to the ABI as depicted in Tables 1 and 2.

Notes:

X0 is always zero,

Table : Integer Register Naming Convention

|  |  |  |  |
| --- | --- | --- | --- |
| Name | ABI Mnemonic | Meaning | Preserved across calls? |
| x0 | zero | Zero | — (Immutable) |
| x1 | ra | Return address | No |
| x2 | sp | Stack pointer | Yes |
| x3 | gp | Global pointer | — (Unallocatable) |
| x4 | tp | Thread pointer | — (Unallocatable) |
| x5 - x7 | t0 - t2 | Temporary registers | No |
| x8 - x9 | s0 - s1 | Callee-saved registers | Yes |
| x10 - x17 | a0 - a7 | Argument registers | No |
| x18 - x27 | s2 - s11 | Callee-saved registers | Yes |
| x28 - x31 | t3 - t6 | Temporary registers | No |

Table : Floating-point register convention

|  |  |  |  |
| --- | --- | --- | --- |
| Name | ABI Mnemonic | Meaning | Preserved across calls? |
| f0 - f7 | ft0 - ft7 | Temporary registers | No |
| f8 - f9 | fs0 - fs1 | Callee-saved registers | Yes\* |
| f10 - f17 | fa0 - fa7 | Argument registers | No |
| f18 - f27 | fs2 - fs11 | Callee-saved registers | Yes\* |
| f28 - f31 | ft8 - ft11 | Temporary registers | No |

Table : UIdaho Pointer Registers

|  |  |  |  |
| --- | --- | --- | --- |
| Name | ABI Mnemonic | Meaning | Preserved across calls? |
| p0 – p7 | p0 – p7 | Collection Pointer Register | No |
|  |  |  |  |

## Register Expansions

Each register is associated with a “Type Tag” as defined in Section 2.

The purpose of the type tag is to specify a data type for the data currently stored in the register. Instruction will be modified (Section 3) and new instructions will be added (Section 4) to support this approach. New exceptions are introduced (Section 5) to support processing errors found due to the type tagging.

## Memory Expansions

Each byte of memory will also be associated with a Type Tag” as Defined in Section 2.

# Type Tags



Figure . Type Tag Layout

## We have the following base types:

* **Raw (Bit Vector)** – these are raw bytes and can be used for a wide collection of representations, for example graphics, sound, etc. (8, 16, 32 or 64 bit).
* **Unsigned integer** – Unsigned numbers. There is debate about the use of these, but they are essential with communication protocols and remote devices; also use for array indices and addresses. (8, 16, 32 or 64 bit)
* **Signed Integer** – Numbers stored in two’s complement representation. (8, 16, 32 or 64 bit)
* **Float** – IEEE 754 Floating point (32 or 64 bit)
* **Code -** normal executable code
* **Code Function Entry** – a function call site.
* **Code Indirect Jump Target** – an instruction that is the target of a jump from an indirect value (for example a switch statement may calculate which case to jump to from a table store that result in a register and jump to the contents of that register).
* Code Return Address – an instruction that follows a call, and is the intended return address of a call.
* **Reference**—A reference to a single entity of the specified type. Size of the stored data is default machine address size (32 bit), references an entity of a specific type.
* **Collection Reference**  - A reference to the first address of a collection of entities of the specified type.
* **Collection Size**- the stored data (1-4 bytes) indicates the number of bytes in a collection (or number of entities?)

The tag format consists of four fields (see Figure 1 and Table 4)

* **SIZE** – indicates the number of additional bytes in the data type. 0, 1, 3, or 7 for a total of 1, 2, 4 or 8 bytes. (00= 0, 01 = 1, 10 = 3, 11 = 7)
* **TYPE** –specifies the base data type according to
  + Special = 000 Code = 001
  + **Collection** = 010 **RAW** (Bit Vector) = 011
  + Unsigned INT = 100 Signed INT = 101
  + Float = 110 Reference = 111
* **REF**  - is used for reference and collection types. The type field indicates what type is being referenced. Note that we have Reference and collections as a type, this allows for “pointers to pointers” and other nested data structures. Size and field is set to default number of bytes for addressing in processor, but is actually ignored.
  + **NONE** = 00  **Base Type Reference =** 01
  + Collection Size = 10 Collection Reference = 11
* **F/MB** – the F/MB bit (first or more bit) is not set (0) if this is the first byte of a data type, it is set (1) if this is a subsequent byte in a multibyte data type. For now, in mmultibyte, the remaining bits are the same as in the first byte.

Notes:

* For now we assume all instructions are 32 bits and use the size bits to specific control flow targets.
* Reference and Collection base types by default are the size of the base address in the processor (32 bits) so size field is ignored.
* A normal type bits 5&6 are 00
* For reference to a base type. Bits 5 & 6 are “01”.
* For a reference to a collection of multiple entities, Bits 5 & 6 are “11”.

NOTE:

* How do I know we are at the start of a collection? Do I ever?
* Check collection size has enough information – is it always unsigned int type?
* How do we handle structs?
* How do we handle arrays of Structs / arrays of collections?

How do we handle arrays of references?

Table . Type Tag Sizes

|  |  |
| --- | --- |
|  | Bits 0-1 |
| Type Size 8 bits | 0 |
| Type Size 16 bits | 1 |
| Type Size 32 bits | 2 |
| Type Size 64 bits | 3 |
|  |  |
|  | Bits 2-4 |
| Special Tag Types | 000 |
| Base Type Code (size 0) | 001 (Need to overload these by using size bits) |
| Base Type Code Function Entry (size 1) | “ |
| Base Type Code Indirect Jump Target (size 2) | “ |
| Base Type Code Return Location (size 3) | “ |
| Base Type Collection | 010 |
| Base Type Raw (Bit vector) | 011 |
| Base Type Unsigned Int | 100 |
| Base Type Signed Int | 101 |
| Base Type Float | 110 |
| Base Type Reference | 111 |

Table : Tag Encodings

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Type Tag for First Byte | Decimal | Type Tag for Subsequent Bytes (if any) |
| NO\_TYPE (no size) | 0 00 000 00 | 0 | This is registers and unallocated memory. |
| ZERO\_TYPE (no size) | 0 00 000 01 | 1 | This is for register Zero |
| VOID\_T (no size) | 0 00 000 10 | 2 | This is registers only |
| UNDECLARED\_T (no size) | 0 00 000 11 | 3 | This is registers only |
|  |  |  |  |
| Code | 0 00 001 00 | 4 | 1 00 100 00 |
| Code Function Entry | 0 00 001 01 | 5 | 1 00 101 01 |
| Code Indirect Jump Target | 0 00 001 10 | 6 | 1 00 101 10 |
| Code Return Location | 0 00 001 11 | 7 | 1 00 101 11 |
|  |  |  |  |
| Collection | 0 00 010 00 | 9 | 1 00 111 00 |
| Collection Size | 0 00 010 01 | 10 | 1 00 111 01 |
|  |  |  |  |
| RAW 8 bits | 0 00 011 00 | 12 |  |
| RAW 16 bits | 0 00 011 01 | 13 | 1 00 011 01 |
| RAW 32 bits | 0 00 011 10 | 14 | 1 00 011 10 |
| RAW 64 bits | 0 00 011 11 | 15 | 1 00 011 11 |
|  |  |  |  |
| Unsigned Integer 8 bits | 0 00 100 00 | 8 |  |
| Unsigned Integer 16 bits | 0 00 100 01 | 9 | 1 00 100 01 |
| Unsigned Integer 32 bits | 0 00 100 10 | 10 | 1 00 100 10 |
| Unsigned Integer 64 bits | 0 00 100 11 | 11 | 1 00 100 11 |
|  |  |  |  |
| Signed Integer 8 bits | 0 00 101 00 | 12 |  |
| signed Integer 16 bits | 0 00 101 01 | 13 | 1 00 101 01 |
| Signed Integer 32 bits | 0 00 101 10 | 14 | 1 00 101 10 |
| Signed Integer 64 bits | 0 00 101 11 | 15 | 1 00 101 11 |
|  |  |  |  |
| Float 32 bits | 0 00 110 10 | 18 | 1 00 110 10 |
| Float 64 bits | 0 00 110 11 | 19 | 1 00 110 11 |
|  |  |  |  |
| Reference | 0 00 111 00 | 24 | 1 00 111 00 |

# Instruction Modifications

## Control Transfer Instructions

Consider the following control transfer instructions and how they are modified for the UIdaho Typed Assembly.

### Unconditional Jumps (JAL,JALR, or pseudo-instruction J)

Where pc is current program counter, j is a standard jump, jal is a jump (call) where return address is stored in rd, conventions makes this x1, ABI named ra (an alternate is x5). We have divided these into the following. For near jump and call (jal) imm is 20 bits and is sign extended to allow forward and backward jumps. For far jump and call (jlar) imm is 12 bits and is sign extended and added to contents of rs1. It is not multiplied by 2.

|  |  |  |
| --- | --- | --- |
| Instruction | Description | |
| jal rd, offset | Jump to pc+ 2\* offset  Store return address in rd | |
| jalr rd, rs1,offset | Jump to pc + (rs1) + imm  Store return address in rd | |
|  |  |  |
|  |  |  |
| **Official Pseudoinstruction** | **Actual Instruction** | **Description** |
| j offset | jal x0, offset | Jump |
| jal offset | jal x1, offset | Jump and link (near call) |
| jr rs | jalr x0, rs, 0 | Jump register (indirect jump) |
| jalr rs | jalr x1, rs, 0 | Jump and link register (call with function pointer) |
| ret | jalr x0, x1, 0 | Return from subroutine |
| call offset | auipc x1, offset[31:12] jalr x1, x1, offset[11:0] | Call far-away subroutine |
| tail offset | auipc x6, offset[31:12] jalr x0, x6, offset[11:0] | Tail call far-away subroutine |

Notes:

* rs1 is usually calculated using LUI or AUIPC
* prefer to use the specified pseudoinsrtuctions
* Tail call occurs when last instruction of a function is a call to another function. We can return from that called function to the parent who called us. Useful when we have multiple nested function calls.
* We want to avoid direct use of jalr with a non-zero offset, unless it is call or tail. Hmm.

Exceptions:

* Instruction-Address-Misaligned -- if target address is not aligned to a four-byte boundary for any of the above instructions.
* Invalid-Function-Call-Address -- If tag of memory address of jal, jalr, call or tail call is not Code Function Entry Point.
* Invalid-Indirect-Address -- If tag of memory address of jr instruction Code Indirect Jump Address.
* Invalid-Return-Address -- If tag of memory address of ret not Code Return Address.
* Type-Mismatch – If tag of rs is not code address? Maybe start with signed integer.

Typing Rules:

* rd (x1) will be tagged as a Code Return Address after successful completion of the instruction.
* rs should be a code address

What about call far-away?

### Conditional Branches (BEQ/BNE/BLT/BLTU/BGE/BGEU)

Conditional branches will calculate a new pc pnly if the comparison of the two source registers is true. The imm is 12 bits and is sign extended to allow forward and backward branches.

bxx rs1 rs2 imm pc ← pc ± imm\*2 if condition true,else pc ← pc + 4

Exceptions:

* Instruction Address Misaligned (0) -- if target address is not aligned to a four-byte boundary for any of the above instructions.

## Computation Instructions

Consider the following integer computation instructions and how they are modified for the UIdaho Typed Assembly.

### Integer Register-Immediate Instructions (Arithmetic) ADDI/SLTI[u]

This performs basic arithmetic with a register and an immediate. Format is:

op rd, rs1, imm

Notes

* The imm is signed 12 bits.

Exceptions:

* Integer Type mismatch (1024) -- If the tag of source register is not an Integer or Zero type.
* Integer overflow (1025) – If the result of the computation exceeds the maximum/minimum value of the type specified in the type tag of rs1. (Only possible with ADDI)

Typing Rules:

* If rs1 is an Integer type, the type tag of rd will be the tag of rs1.
* If rs1 is Zero type, the type tag of rd will be the smallest Integer type to hold the value?
* Computation will be performed as if all values are 32bits. With imm sign extended.

### Integer Register-Immediate Instructions (Logical) ANDI/ORI/XORI

This performs basic bitwise operations with a register and an immediate. Format is:

op rd, rs1, imm

Notes

* The imm is signed 12 bits.

Exceptions:

* Integer type mismatch (1024) -- If the tag the source register is not a RAW or Zero type.

Typing Rules:

* If rs1 is an Integer type, the type tag of rd will be the tag of rs1.
* If rs1 is Zero type, the type tag of rd will be the smallest RAW type to hold the value?
* Computation will be performed as if all values are 32bits. With imm sign extended.

### Integer Register-Immediate Instructions (Shifting) SLLI/SRLI/SRAI

This performs basic bitwise shifting operations with a register and an immediate. Format is:

op rd, rs1, imm

**Notes**

* The imm is unsigned 5 bits.

**Exceptions**:

* Integer type mismatch (1024) -- If the tag the source register is not a RAW type.
* Integer type mismatch (1024) – For SLLI and SRLI if the tag of the source register is not unsigned int
* Integer type mismatch (1024) – For SRAI if the tag of the source register is not integer
* Integer overflow (1025) – If the tag type is integer and the result exceed the specified bit width.

**Typing Rules:**

* The type tag of rd will be the tag of rs1.
* Computation will be performed as if rs1 is 32bits.
* ?? If bit width of tag of rs1 < 32 bits, all upperbits will be sign extended from nth bit?

**Note:**

* For SLL, the operation needs to be aware of bit width and set the appropriate upper bits (?)
* This is difficult in hardware, so we may want to avoid it.

### Integer Register-Immediate Instructions (Constants) LUI/AUIPC

These are used to build 32-bit constants. Sometimes in conjunction with indirect addressing. To build a full 32-bit constant with LUI you use LUI to set upper 20 bits, then ADDI to set remaining lower 12 bits. (If addi’s immediate most significant bit is a 1, it sign extends, so we need to be careful. We solve that by adding a 1 to the value for the upper 20 bits first.

For AUIPC we are building PC relative addresses. Usually just the upper 20 bits, then add in the lower 12 immediate from JALR

op rd, imm

Notes:

* The imm is unsigned 20 bits

Exceptions:

Typing Rules:

* The type tag of rd is will be integer type for LUI and code indirect target type for AUIPC.

Note:

* We will never construct an address for other code types.

## Integer Register-Register Operations

### Integer Register-Register Instructions (Arithmetic) ADD/SUB/SLT[u]

This performs basic arithmetic with two registers. Format is:

op rd, rs1, rs2

Exceptions:

* Integer type mismatch (1024) -- If the tag of two source registers are not the same (one can be Zero type, but not both).
* Integer type mismatch (1024) -- If the tag of two source registers are not integers (one can be Zero type).
* Integer overflow (1025) – If the result of the computation exceeds the maximum/minimum value of the type specified in the type tag of rs1. (Only possible with ADD/SUB)

Typing Rules:

* If the tag of rs1 is not Zero type, the type tag of rd will be the tag of rs1.
* If the tag of rs1 is Zero type, the type tag of rd will be the tag of rs2.
* Computation will be performed as if all values are 32bits. With imm sign extended.

### Integer Register-Register Instructions (Logical) AND/OR/XOR

This performs basic logical operations between with two registers. Format is:

op rd, rs1, rs2

Exceptions:

* Integer type mismatch (1024) -- If the tag of two source registers are not the same (one can be Zero type, but not both).
* Integer type mismatch (1024) -- If the tag of two source registers are not RAW type (one can be Zero type).

Typing Rules:

* If the tag of rs1 is not Zero type, the type tag of rd will be the tag of rs1.
* If the tag of rs1 is Zero type, the type tag of rd will be the tag of rs2.
* Computation will be performed as if all values are 32bits. With imm sign extended.

### Integer Register-Register Instructions (Shifting) SLL/SLR/SRA

This performs basic shifting operations between with two registers. Format is:

op rd, rs1, rs2

Exceptions:

* Integer type mismatch (1024) -- If the tag the rs1 is not a bitwise or integer.
* Integer type mismatch (1024) – For SLL and SRL if the tag of rs1 not unsigned int.
* Integer type mismatch (1024) – For SRAI if the tag of rs1 is not integer.
* Integer type mismatch (1024) -- If the tag the rs2 is not unsigned int 8
* Integer overflow (1025) – If the tag type of rs1 is integer and the result exceed the specified bit width.

Typing Rules:

* The type tag of rd will be the tag of rs1.
* Computation will be performed as if rs1 is 32bits.
* If bit width of tag of rs1 < 32 bits, all upperbits will be set to 0.

Note:

* For SRAI, the operation needs to be aware of bit width and set the appropriate upper bit.
* This is difficult in hardware, so we may want to avoid it.

## Memory Access Instructions

### Memory Access Instructions (lb, lbu, lh, lhu, lw)

This performs memory load using indirect addressing. One method is rs1 contains a stack or frame pointer and imm is the offset. Another is rs1 may be a global pointer to a global table and imm is the offset into that table. LB loads an 8 bit value and sign extends into the 32 bits. LBU loads an 8 bit and zero extends. LH and LHU are similar for 16 bit words and LW loads a 32 bit. We assume we

Format is

op rd, rs1, imm

Exceptions:

* Tagged load address misaligned (1030) - If the tag of the target does not match the start of a data value that matches the requested load size/type.
* Tagged load nondate (1031) – If the target of the load is not a data type:
* Tag type (bits 2-4) 2-4 = 0
* Tagged load untagged (1032) – If the target of the load is not tagged (tag 0).

Typing Rules:

All memory is tagged for values stored in that location.

# New Instructions

## Type Conversion

## Pointer Manipulation

# New Exceptions

We need to support precise exceptions for the errors

Table 6: Exceptions

|  |  |  |
| --- | --- | --- |
| Interrupt | Exception Code | Description |
| 0 | 0 | Instruction address misaligned |
| 0 | 1 | Instruction access fault |
| 0 | 2 | Illegal instruction |
| 0 | 3 | Breakpoint |
| 0 | 4 | Load address misaligned |
| 0 | 5 | Load access fault |
| 0 | 6 | Store/AMO address misaligned |
| 0 | 7 | Store/AMO access fault |
| 0 | 8 | Environment call from U-mode |
| 0 | 9 | Environment call from S-mode |
| 0 | 10 | Reserved |
| 0 | 11 | Environment call from M-mode |
| 0 | 12 | Instruction page fault |
| 0 | 13 | Load page fault |
| 0 | 14 | Reserved |
| 0 | 15 | Store/AMO page fault |
| 0 | ≥16 | Reserved |
| The Following are for UI Exceptions (Set bit | | |
| 0 | 1024 | Integer type mismatch |
| 0 | 1025 | Integer overflow |
| 0 | 1026 | Integer divide by zero |
| 0 | 1032 | Invalid function call address |
| 0 | 1033 | Invalid indirect address |
| 0 | 1034 | Invalid return address |
|  | 1034 | Code type mismatch |
|  | 1040 | Tagged load address misaligned |
|  | 1041 | Tagged load address not data |
|  | 1042 | Tagged load address untagged |
|  |  |  |

# Experimentation

To support evaluation and experimentation for this system, we implement the following new instructions. These will not be made available in the production version of the new processor.

## Tag Control

To enable focused testing, we introduce instructions to turn on and off tag checking. We support turning on and off tag propagation, and separately turning on and off tag checking. This allows for fine-grain evaluation of the enhanced model.

We use the RV31I custom hints for the encode. A hint is a NOP in the standard RSICV ISA (usually the target register is x0 which is immutable). Hints are meant to allow customized performance hints for the processor. This allows us to modify assembly code to include these instructions without breaking any of the standard tools.

### Tag Propagation Controls

|  |  |  |
| --- | --- | --- |
| Mnemonic | Meaning | Encoding |
| TPN | Turn Tag Propagation Off | SLTI x0, x1, 0 |
| TPY | Turn Tag Propagation On | SLTI x0, x1, 1 |
| STT | Set a Tag | SLT x0, rd1, rs2 |
| RDT | Read a Tag | SLL x0, rd1, rs2 |

When tag propagation is on, all of the new tag propagation rules specified for instructions are implemented, but no checks are made to test for tag propagation errors.

Set a tag (STT) provides the programmer with the ability to copy a tag, currently stored in register rs2, into the tag portion of register rs1.

Read a tag (RDT) provides the programmer with the ability to copy a tag, currently stored in register rs2, into the tag portion of register rs1.

The set/read tag operations allow us to set up tag values for experimentation. The normal process is to:

* set tags of several registers
* turn on tag propagation (and maybe checking)
* execute the experiment code
* turn off tag propagation (and checking)
* maybe: copy tags to standard registers and then call print functions to display results.

**NOTE:** No additional tag checking or tag propagation is performed by these operations.

### Tag Checking Controls

|  |  |  |
| --- | --- | --- |
| Mnemonic | Meaning | Encoding |
| TCN | Turn Tag Checking Off | SLTI x0, x1, 2 |
| TCY | Turn Tag Checking On | SLTI x0, x1, 3 |
| TBN | Turn Tag Checking and Propagation Off | SLTI x0, x1, 4 |
| TBY | Turn Tag Checking and Propagation On | SLTI x0, x1, 5 |
| TCNI | Turn Tag Checking Off for a specific exception | SLTIU x0, x1, EXN |
| TCYI | Turn Tag Checking On for a specific exception | SLTIU x0, x1, EXN |

When tag checking is on, any violation of the tag checking rules will result in throwing of an exception.

To allow for more precise testing, we allow for turning on and off select exceptions (using the exception numbers specified in Section 5).

For example, we can turn on the exceptions for control flow integrity checks without full implementation of buffer overflow checks.

**NOTE:** No additional tag checking or tag propagation is performed by these operations

# Notes for Emulator Expansion

If we have an existing RISCV emulator that we wish to modify, we will need to do the following:

1. Expand registers to store tags. We need to store the tag values associated with each register. The following is example for SPIKE
   1. Registers are implemented in a regfile class, with a data field.
   2. STATE.XPR.write(reg, wdata) is used to set the data value of general purpose register “reg”
   3. There is an operator [] to read register values (such as XPR[3])
   4. To modify:
      1. Add a class field for tag.
      2. Add new method to read the tag.
      3. EModify write to take an additional parameters to set the tag (maybe default it to the UNDEFINED tag value if not provided)
      4. Seach all of the simulator code to update the setting/reading of tags here are some hints:

look in decode.h

-- definex NXPR, NFPR, NVPR register counts

-- defines regfile\_t (this is where we add typing)

look in decode\_macros

-- uses XPR[reg] and FPR[reg]

look in processor.h

-- defines state

-- incldues XPR, FPR

decode\_macros.h has START.XPR.write

rocc.h

-- start->XPR.write

-- uses XPR[

interactive.cc ??

-- uses XPR[

-- uses FPR[

1. Add instructions to turn on and off tag propagation and checking.
   1. Add a field to STATE to represent the statis of tagging.
   2. In SPIKE we normally do the following:
      1. Describe the instruction's functional behavior in the file riscv/insns/<new\_instruction\_name>
      2. Add the opcode and opcode mask to riscv/opcodes.h
      3. Rebuild the simulator
      4. You can examine other instructions in the riscv/insns directory for a starting point.
   3. However in SPIKE. might just need to change behavior of parent instruction for our hints. For example look at the more complex code in insns/rem.h and insns/wfi.h
2. Add instructions to set/read tags
   1. Modify the instructions like above.
   2. Now run some code to test to see if this works just set/read.
3. Modify a single instruction to support either tag propagation and or checking.
   1. Test it using the above
   2. Do this one instruction group at a time (as defined in the instructions section
      1. Simple math operations
      2. Control Flow
      3. Load/Store
      4. …

## HOW TO TEST

I have created a directory (asmTest) that will be a subdirectory in FreeRTOS

…FreeRTOS/FreeRTOS/tests/asmTest

If you type “make” then

spike -p1 --isa RV32IMA -m0x80000000:0x10000000 --rbb-port 9824 \ ./build/RTOSDemo32.axf

You will see results of a Hello World program.

Change line 10 of main.S to get a new output message. (make and run spike)

Insert new commands after line 20 to test

-- See notes in UidahoAsm.txt